

Programmation Web Avancée

Cours 2

Objets

Portée des variables

Tableaux

Rappels MVC



kn@lri.fr



Plan

- 1 Introduction/Généralités et rappels sur le Web/Javascript : survol du langage ✓
- 2 Objets/Portée des variables/Tableaux/Rappels MVC
 - 2.1 Objets
 - 2.2 Portée
 - 2.3 Tableaux
 - 2.4 MVC

Principes de la programmation orientée objet

Un général, un *langage orienté objet statiquement typé* propose une notion de **classe** et d'**objet**. Par exemple en Java :

```
class Point {  
    private int x;  
    private int y;  
  
    Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public void move(int i, int j) {  
        this.x += i;  
        this.y += j;  
    }  
    public int getX() { return this.x; }  
    public int getY() { return this.y; }  
}
```

Une **classe** définit un ensemble d'**objets** contenant un état interne (les attributs : **x**, **y**) ainsi que du code (les méthodes : **move**, ...) permettant de manipuler cet état. Un objet est l'**instance** d'une classe.

Concepts objets

Les langages orientés objets exposent généralement plusieurs concepts :

- ◆ La notion de **constructeur** : une fonction ayant un statut spécial, qui est appelé pour **initialiser** l'état interne de l'objet à sa création.
- ◆ Une notion de **contrôle d'accès** aux attributs et méthodes.
- ◆ Un moyen de **référencer** l'objet dans lequel on se trouve (**this**).
- ◆ Plusieurs notions permettant de partager du code ou des données (**static**, héritage, ...)

⇒ Javascript ne fait pas de différences entre « **attributs** » et « **méthodes** ».

⇒ Les « champs » d'un objet sont appelés « **propriétés** ». Elles peuvent contenir des valeurs scalaires ou des fonctions.

⇒ L'affectation à une propriété en Javascript **ajoute** la propriété si elle était inexistante :

```
let p1 = { };           //Un objet vide
p1.x = 0;               //On ajoute un champ x initialisé à 0
p1.y = 0;               //On ajoute un champ y initialisé à 0

//Ajout de « méthodes » move, getX et getY
p1.move = function (i, j) { p1.x += i; p1.y += j; };
p1.getX = function () { return p1.x; };
p1.getY = function () { return p1.y; };
```

Quels sont les problèmes avec le code ci-dessus ?

1. Il faut copier-coller **tout** le code si on veut créer un autre point **p2**
2. Pour chaque objet **p_i**, on va allouer 3 fonctions **différentes** (qui font la **même** chose pour l'objet considéré.)

5 / 35

Function, prototype et new

En Javascript, le type « **Function** » (des fonctions) a un statut particulier. Lorsque l'on appelle l'expression **new f(e₁, ..., e_n)** :

1. Un nouvel objet **o** (vide) est créé.
2. Le champ **prototype** de **f** est copié dans le champ **prototype** de **o**.
3. **f(e₁, ..., e_n)** est évalué et l'identifiant spécial **this** est associé à **o**
4. Si **f** renvoie un **objet**, alors cet objet est le résultat de **new f(e₁, ..., e_n)**, sinon l'objet **o** est renvoyé.

L'expression **new e** où **e** n'est pas un appel de fonction provoque une **erreur**.

Comment créer des objets avec ça ?

7 / 35

On englobe le tout dans une **fonction** :

```
let mkPoint = function(x, y) {
  let p = { };
  p.x = x;
  p.y = y;
  p.move = function (i, j) { p.x += i; p.y += j; };
  p.getX = function () { return p.x; };
  p.getY = function () { return p.y; };
  return p;
};

...
let p1 = mkPoint(1,1);
let p2 = mkPoint(2, 10);
let p3 = mkPoint(3.14, -25e10);
...
```

La fonction **mkPoint** fonctionne comme un **constructeur**. Cependant, les trois « méthodes » sont allouées **à chaque fois**.

6 / 35

Function, prototype et new (suite)

```
let Point = function (x, y) {
  this.x = x;
  this.y = y;
};

...
let p1 = new Point(1,1);
let p2 = new Point(2, 10);
let p3 = new Point(3.14, -25e10);
...
```

1. Un nouvel objet **p₁** (vide) est créé.
2. Le champ **prototype** de **Point** est copié dans le champ **prototype** de **p₁**.
3. **Point(e₁, ..., e_n)** est évalué et l'identifiant spécial **this** est associé à **p₁**
4. Si **Point** renvoie un **objet**, alors cet objet est le résultat de **new Point(e₁, ..., e_n)**, sinon l'objet **p₁** est renvoyé.

8 / 35

La *résolution de propriété* en Javascript suit l'algorithme ci-dessous. Pour rechercher la propriété **p** sur un objet **o** :

```
objet_courant ← o;
répéter:
  si objet_courant.p est défini alors renvoyer objet_courant.p;

  si objet_courant.prototype est défini, différent de null et est un objet,
    alors objet_courant ← objet_courant.prototype
```

Une propriété **p** d'un objet **o** peut donc être :

- ◆ Rattachée à **o** lui-même (**p** est alors une *propriété propre* de **o**, *own property*)
- ◆ Ou rattachée à l'objet **o₁** se trouvant dans le champ **prototype** de **o** (s'il existe)
- ◆ Ou rattachée à l'objet **o₂** se trouvant dans le champ **prototype** de **o₁** (s'il existe)
- ◆ ...

9 / 35

Parallèle avec les langages compilés

Le fonctionnement par **prototype** est identique à la manière dont les langages OO statiquement typés (Java, C++, C#) sont **compilés**.

En Java, chaque objet contient un pointeur (caché) vers un *descripteur de classe* (une structure contenant les adresses de toutes les méthodes de la classe + un pointeur vers le descripteur de la classe parente) = **prototype**.

Qu'offre Javascript en plus ?

- ◆ Redéfinition **locale** de propriétés (*monkey patching*)
- ◆ Définition manuelle du prototype pour « hériter » d'un type existant

11 / 35

```
let Point = function (x, y) {
  this.x = x;
  this.y = y;
};
Point.prototype.move = function (i, j) {
  this.x+= i;
  this.y+= j;
};
Point.prototype.getX = function () {
  return this.x;
};
Point.prototype.getY = function () {
  return this.y;
};
...
let p1 = new Point(1,1);
p1.move(2, 10);
...
```

Lors de l'appel à **move** l'objet **p1** ne possède pas directement de propriété **move**. La propriété **move** est donc cherchée (et trouvée) dans son champ **prototype**.

10 / 35

Monkey patching

Technique qui consiste à redéfinir une méthode sur un **objet spécifique** (impossible à faire en Java).

```
let p1 = new Point(1, 1);
let p2 = new Point(1, 1);

p2.move = function () { this.x = 0; this.y = 0;};
p1.move(10, 10); //appelle Point.prototype.move
p2.move(10, 10); //appelle la méthode move définie ci-dessus

let x1 = p1.getX(); //x1 contient 11
let x2 = p2.getX(); //x2 contient 0
```



: c'est une technique dangereuse, car elle donne un comportement **non-uniforme** à des objets du même « type ». On l'utilisera à des fins de debuggages, jamais pour spécialiser durablement le type d'un objet (et encore moins d'un objet système tel que **Math**).

12 / 35

On peut savoir à tout moment si un objet `o` a une propriété `p` propre en utilisant la méthode `.hasOwnProperty(...)`

```
let p = new Point(1, 2);
p.hasOwnProperty('x'); // renvoie true
p.hasOwnProperty('move'); // renvoie false
```

13 / 35

Opérateur instanceof

En Javascript, l'opérateur `instanceof` existe et permet de tester si un objet est bien d'une certaine «classe». Il applique l'algorithme suivant

```
e instanceof C

si C n'est pas une fonction, FAUX
evaluer e en o. Si o n'est pas un objet, renvoyer FAUX. Sinon :
P ← C.prototype
Si P n'est pas un objet, erreur de typage. Sinon :
répéter :
  o ← o.prototype
  si o est null, renvoyer FAUX
  si P ≡ o, renvoyer VRAI
```

En d'autre termes, l'opérateur `instanceof` remonte la chaîne des prototypes jusqu'à trouver le même que celui de l'objet passé en argument ou trouver `null` (car on est remonté jusqu'à `Object`).

15 / 35

L'algorithme de résolution de propriété peut être utilisé pour simuler l'héritage.

```
let ColoredPoint = function (x, y, c) {
  Point.call(this, x, y); //appel du constructeur parent
  this.color = c || "black"; //si c est convertible en false
                             //on initialise à black
};
ColoredPoint.prototype = Object.create(Point.prototype);
//Object.create crée une copie de l'objet passé en argument
//(on peut lui passer d'autres paramètres).

ColoredPoint.prototype.constructor = ColoredPoint;
//On met à jour le champ constructor, qui vaut encore Point

ColoredPoint.prototype.getColor = function () { return this.color; };

let p = new ColoredPoint(1, 2, "red");
p.move(10, 10); //move se trouve dans ColoredPoint.prototype.prototype !
p.getColor(); //getColor se trouve dans ColoredPoint.prototype !
```

14 / 35

Définition de propriété

Une alternative à la définition simple de propriété (`o.x = v`) est l'utilisation de la fonction `Object.defineProperty(obj, prop, conf)`. Cette fonction ajoute à l'objet `obj`, la propriété de nom `prop`. L'objet `conf` permet d'ajuster finement les caractéristiques de la propriété :

value : la valeur de cette propriété
writable : Si `false`, la mise à jour de la propriété sera sans effet
get : Une fonction sans argument qui renvoie la valeur de la propriété
set : Une fonction à un argument qui met à jour la valeur de la propriété

```
let o = { };
Object.defineProperty(o, "x", { value : 5, writable : false });
Object.defineProperty(o, "y", { get : function () { return Math.random (); },
                                set : function (x) {} });

o.x = 3;
o.x // 5
o.y // 0.1231455739
o.y = 3;
o.y // 0.8467374344
```

16 / 35

Il existe trois niveaux de protection des objets :

`Object.preventExtension(o)` : Impossible d'ajouter de nouvelles propriétés *propres* possibilité de supprimer (avec *delete*) une propriété propre existante. Impossibilité d'utiliser `Object.setPrototype`.

`Object.seal(o)` : Comme `Object.preventExtension(o)` avec en plus l'impossibilité de supprimer des propriétés.

`Object.freeze(o)` : Comme `Object.seal(o)` avec en plus l'impossibilité de *modifier* des propriétés (l'objet devient *read-only* ou immuable)

On peut tester l'état d'un objet avec les méthodes `Object.isExtensible/.isSealed/.isFrozen`. Il est impossible de revenir en arrière

17 / 35

Plan

1 Introduction/Généralités et rappels sur le Web/Javascript : survol du langage ✓

2 Objets/Portée des variables/Tableaux/Rappels MVC

2.1 Objets ✓

2.2 Portée

2.3 Tableaux

2.4 MVC

```
class C extends Parent {
  constructor (x, y, z) {
    super (y, z);           //appel du constructeur parent
    this.prop1 = ...;
    ...
  }

  f (x, y, z) {             //methode ajoutée au prototype
    ...
  }

  static g (x, y, z) {      //methode ajoutée au constructeur
  }

  get prop () {             //definit un getter pour o.prop
  }
}
```

Attention ! pas du tout supporté par IE, dans Edge à partir de la version 13 uniquement (Windows 10).

18 / 35

Objet Global et variables globales

La norme Javascript (ECMA-262) définit un **Objet Global** initialisé avant le début du programme.

Les variables globales en Javascript ne sont que des *propriétés propres* de cet objet.

Dans les navigateurs Web, cet objet global représente l'«onglet courant». Il possède une propriété **window** qui pointe sur lui même.

```
//On suppose que l'on est dans un fichier test.js inclus directement
//dans la page
```

```
let foo = 123;           // variable « globale »
bar = 456;               // assigne la propriété bar à l'objet global
this.baz = 789;          // idem mais avec un this explicite
```

```
window.foo;             // vaut 123
window.bar;             // vaut 456
window.baz;             // vaut 789
```

var, c'est le mal

En Javascript < 5, la seule construction pouvant introduire une nouvelle portée (*scope*) est **function** (...) {}.

Une variable déclarée (au moyen de **var**) dans une fonction est *locale* à cette fonction.



les « blocs » ne créent pas de nouvelle portée!

```
//On suppose que l'on est dans un fichier test.js inclus directement
//dans la page
for(var i = 0; i < 10; i++) {
  var j = 2 * i;
  ...
}
i;           //bien défini après la boucle, vaut 9
j;           //bien défini après la boucle, vaut 18
var f = function () {
  var k = 20;
  ...
};
k;           //vaut undefined
```

21 / 35

var, c'est toujours le mal

Les déclarations de variables *locale* sont déplacées (*hoisted*) en début de portée. Ainsi :

```
function f () {
  console.log('Hello !');
  var x = 23;
  ...
};
```

est équivalent à :

```
function f () {
  var x;
  console.log('Hello !');
  x = 23;
  ...
};
```

Rappel : var est proscrit des TPs, du projet et de l'examen

23 / 35

Shadowing

Une variable peut masquer une variable de même nom se trouvant dans une portée englobante:

```
//On suppose que l'on est dans un fichier test.js inclus directement
//dans la page
let x = 123;
console.log(x);           // affiche 123
function f () {
  let x = 456;
  function g () {
    let x = 789;
    console.log(x);       // affiche 789 quand g est appelée
  };
  g ();
  console.log(x);         // affiche 456 quand f est appelée
};
f ();
console.log(x);           // affiche 123
```

22 / 35

Identifiant this

L'identifiant **this** est similaire à celui de Java **mais** est **tout le temps défini**, avec des règles précises :

- ◆ o.f(e₁, ..., e_n) : **this** est initialisé à **o**
- ◆ new f(e₁, ..., e_n) : **this** est l'objet fraîchement créé.
- ◆ f.call(o, e₁, ..., e_n) : **this** est initialisé à **o**
- ◆ f(e₁, ..., e_n) :  **this** est initialisé à l'objet global (**window**).

```
ColoredPoint.prototype.getColor = function () {
  console.log(this.color);           //accès à la couleur
  let g = function () {
    console.log(this.color);         //undefined car équivalent à
                                     //window.color
  };
  g();
  return this.color;
};
```

24 / 35

- 🤖 Le standard ECMAScript définit la notion de modules, d'imports et d'exports
- 🤖 Aucun navigateur ne supporte ça en standard

```
//Définition de Point, move, getX, ...
let Point = function (x, y) {
  ...
};
...
let strAux = function (x, y) {      //on définit une fonction auxiliaire
  return "(" + x + ", " + y + ")";
};

Point.prototype.toString = function () {
  return strAux(x, y);
};

window.strAux (...);           //est défini
strAux(...);                  //est défini
```

Problème : une fonction auxiliaire qui n'est utile qu'à la classe **Point** est visible globalement.

25 / 35

On peut utiliser des fonctions pour encapsuler le code :

```
//Définition de Point, move, getX, ...
let Point = function (x, y) {
  ...
};
...
Point.prototype = (function () {
  //On est maintenant dans une fonction, strAux ne pourra pas
  //s'échapper dans le contexte global
  let strAux = function (x, y) { return "(" + x + ", " + y + ")"; };
  //On renvoie un objet faisant office de prototype :
  return {
    move : function (i, j) { ... },
    getX : function () { return this.x; },
    getY : function () { return this.y; },
    toString : function () { return strAux(x,y); }
  };
})(); //On applique immédiatement la fonction !

strAux(...);                  //undefined
```

26 / 35

Utilisation de fonctions pour encapsuler

- ◆ Ça marche
- ◆ Peut nuire à la lisibilité
- ◆ Incompatible avec la syntaxe de classe

⇒ Il vaut mieux utiliser les modules. On verra comment les utiliser même dans les navigateurs ne les supportant pas

27 / 35

Plan

- 1 Introduction/Généralités et rappels sur le Web/Javascript : survol du langage ✓
- 2 Objets/Portée des variables/Tableaux/Rappels MVC
 - 2.1 Objets ✓
 - 2.2 Portée ✓
 - 2.3 Tableaux
 - 2.4 MVC

Les tableaux (classe **Array**) font partie de la bibliothèque standard Javascript. On peut créer un tableau vide avec **[]**.

```
let tab = [];
tab[35];           //undefined
tab[35] = "Hello"; //initialise la case 35 à "Hello"
tab[0];            //toujours undefined;
tab.length;        //36 ! indice le plus grand ayant été
                  //initialisé + 1
```

Array, interface fonctionnelle

.forEach(f) : Applique la fonction **f** à tous les éléments du tableau qui ne valent pas **undefined**. **f** reçoit trois arguments (**v**, **i**, **t**) :

v : la valeur courante de la case visitée
i : l'indice courant (à partir de 0)
t : le tableau en entier

.map(f) : Applique la fonction **f** à tous les éléments du tableau qui ne valent pas **undefined** et renvoie le tableau des images. **f** reçoit trois arguments (**v**, **i**, **t**), comme pour **forEach**.

.filter(f) : Renvoie le tableau de tous les éléments pour lesquels **f** renvoie vrai. **f** reçoit trois arguments (**v**, **i**, **t**), comme pour **forEach**.

.reduce(f, acc0) : Renvoie l'agrégat de tableau de tous les éléments par **f**. **f** reçoit quatre arguments (**acc**, **v**, **i**, **t**), où **acc** est la valeur courante de l'accumulateur et les autres sont comme pour **forEach**.
 Si **acc0** est absent, exécute l'agrégat entre le premier élément du tableau et le reste.

new Array(n) : Initialise un tableau de taille **n** (indiqué de 0 à **n-1**) où toutes les cases valent **undefined**
.length : renvoie la longueur du tableau
.toString() : applique **.toString()** à chaque élément concatène chaque résultat avec des **","** intermédiaires.
.push(e) : ajoute un élément en fin de tableau
.pop() : retire et renvoie le dernier élément du tableau. **undefined** si le tableau est vide
.shift() : retire et renvoie le premier élément du tableau. **undefined** si le tableau est vide
.unshift(e) : ajoute un élément au début du tableau
.splice(i, n, e1, ..., ek) : à partir de l'indice **i**, efface les éléments **i** à **i+n-1** et insère les éléments **e1, ..., ek**
.concat(t) : Renvoie la concaténation du tableau et du tableau **t**
.sort(f) : Trie le tableau en place. **f** est une fonction de comparaison (**f(a,b)** doit être négatif si **a<b**, nul si **a=b** et positif si **a>b**).

Plan

- 1 Introduction/Généralités et rappels sur le Web/Javascript : survol du langage ✓
- 2 Objets/Portée des variables/Tableaux/Rappels MVC
 - 2.1 Objets ✓
 - 2.2 Portée ✓
 - 2.3 Tableaux ✓
 - 2.4 MVC

Qu'est-ce que le modèle MVC ?



C'est un *design pattern* qui permet de modéliser des applications « interactives » :

- ◆ L'application possède un état interne
- ◆ Un « utilisateur » (ça peut être un programme externe) interagit avec le programme pour modifier l'état interne
- ◆ L'application affiche à l'utilisateur le résultat de son opération

Ces trois aspects sont représentés par trois composants :

- ◆ Le **Modèle** (représentation de l'état interne)
- ◆ La **Vue** (affichage du modèle)
- ◆ Le **Contrôleur** (modification du modèle)

33 / 35

En quoi est-ce adapté aux applications Web ?



Une application Web typique :

- ◆ Présente au client un formulaire permettant de passer des paramètres (**C**)
- ◆ Effectue des opérations sur une base de donnée (**M**) à partir des paramètres
- ◆ Affiche une page Web montrant le résultat de l'opération (**V**)

34 / 35

Avantages du Modèle MVC ?



La **séparation** permet d'obtenir :

Maintenance simplifiée

Le code d'une action est centralisé à un seul endroit

Séparation des privilèges

Pas besoin que la vue ait un accès à la base de donnée par exemple

Test simplifié

Les composants peuvent être testés indépendamment

35 / 35